Research Science Institute 1998, Intel STS 1999

Implementation of Constraint Systems for Useless Variable Elimination

Daniel M. Roy
under the direction of
Prof. Mitchell Wand
Northeastern University

Abstract

Superfluous variables are often produced as the byproducts of program transformations, compilation, and poorly written code. These variables are irrelevant to the computational outcome of their programs. Eliminating them as a means of higher-level optimization may increase program execution speed. This paper explores an implementation of Wand and Siveroni's algorithm for useless variable elimination. The algorithm is shown to remove superfluous variables that are accessed, updated, and passed between functions, as well as collapse `let` expressions when all the `let` expression's variables are superfluous. The algorithm does not preserve non-termination nor remove variables whose contributions are constant.

1 Introduction

Compilation and other program transformations often generate superfluous variables whose values are irrelevant to the final outcome of a computation (Wand and Siveroni, 1999). These variables are termed *useless* (Shivers, 1991). Commercial optimizing compilers use a variety of optimization algorithms, the majority of which perform code streamlining and, consequently, ignore useless variables (Wand, 1998). These compilers optimize only the code, not the algorithms upon which the programs are based. Although these optimizations yield impressive results, higher-level optimization such as dead code elimination and algorithm optimization can dramatically improve program performance (Wand, 1998).

This project deals with an extension of dead code elimination known as *useless variable elimination*. Through control flow analysis and dependency analysis, the useless variable elimination algorithm generates a system of constraints—that is, a series of *subset or equal to* relationship between the sets created from control flow and dependency analysis. These constraints can be solved algebraically to determine which variables are useless not only in each section of the program, but in the entire program (Wand and Siveroni, 1999). Dead code elimination can be more effective than code streamlining; while streamlining will only speed up a complex algorithm, an unreachable code elimination algorithm will remove that algorithm altogether if it determines that the algorithm is simply never run (Muchnick, 1997). The useless variable elimination algorithm determines which calls to the complex algorithm are superfluous, and removes these calls.

This paper details not only the underlying theory, but also the structural composition and testing of the Useless VAriable Removal Program (UVARP), the first implementation of an algorithm for eliminating useless variables presented by Wand and Siveroni (1999). UVARP takes functional Scheme code and returns a new version of the code with the useless variables removed. UVARP was tested on several sample programs to evaluate its performance and illustrate its weaknesses.

2 Control Flow and Dependency Analysis Theory

2.1 Control Flow Analysis

UVARP parses Scheme code, transforming it into the *internal syntax*: an untyped, labeled, lambda calculus. UVARP first assigns a unique numerical label to each Scheme term. The combined unique label and term compose an expression (Nielson and Nielson, 1997). For example, in the expression $(0\ \texttt{fn}\ (x)\ (+\ x\ 1))$, the number, 0, represents the expression's label; the remainder of the expression represents the term. This term is composed of the symbol, `fn`, followed by the arguments of the type, $(x)\ (+\ x\ 1)$.[1] In order to avoid confusion between similarly named variables in different sections of the program, the internal syntax is then alpha converted, a process that assigns a unique name to every unique variable.

After the transformation to internal syntax, UVARP performs Nielson and Nielson's zero-order control flow analysis (0CFA). The algorithm visits each expression and any subexpression within that original expression. Therefore, the 0CFA algorithm recursively steps downward through the entire program. 0CFA generates two sets of information: the *environment* and *configuration*. For each variable *x*, the environment, $\hat{\rho}(x)$, is the set of labels of the possible values of the variable x (Nielson and Nielson, 1997). For each labeled term l, the configuration, $\hat{C}(l)$, is the set of labels of the possible values of label l (Nielson and Nielson, 1997). During the analysis, constraints of the form *subset and equal to* are established between various configurations and environments.

[1] $f(x) = x + 1$

These constraints are algebraically solved, synchronous to the program's analysis.

For example, in Table 2.1, (`lambda` (x) $(add1\ x)$) is an expression in which x is the only variable. $\hat{\rho}(x)$ contains all possible values of x. The possible values for `fn` (x) $(...)$ are the members of the set $\hat{C}(0)$, the possible values for $(1$ `var` $add1)$ are the members of the set $\hat{C}(1)$, and the possible values for $(2$ `var` $x)$ are the members of the set $\hat{C}(2)$.

0CFA analysis uses a universe Σ to map each label to its respective term. Thus, $\Sigma(l)$ is the term labeled l. According to Nielson and Nielson, $(\Sigma, l) \models (\hat{C}, \hat{\rho})$ if $\hat{C}$ and $\hat{\rho}$ satisfy the constraints generated from the definition of $\models$ for the term $\Sigma(l)$ (Nielson and Nielson, 1997). $(\Sigma, l) \models (\hat{C}, \hat{\rho})$ is read as: The expression l is *consistently described* by the sets $\hat{C}$ and $\hat{\rho}$ in universe Σ. $\models$ is defined differently for each syntactical expression supported by UVARP.

For a constant, $(\hat{C}, \hat{\rho})$ always consistently describes (Σ, l). For any variable x labeled l, $(\Sigma, l) \models (\hat{C}, \hat{\rho})$ if and only if $\hat{\rho}(x) \subseteq \hat{C}(l)$. For example, in Table 2.1, $\hat{\rho}(x) \subseteq \hat{C}(2)$. Further definition of control flow analysis may be found in Appendix A.

2.2 Dependency Analysis

While performing 0CFA analysis to determine $\hat{\rho}$ and $\hat{C}$, UVARP conducts dependency analysis to create a set, $\mathcal{D}(l)$, of the set of all variables that contribute to the value of a labeled expressions $\Sigma(l)$ (Wand and Siveroni, 1999). A variable is deemed useless for $\Sigma(l)$ if it does not appear in $\mathcal{D}(l)$ (Wand and Siveroni, 1999).

During dependency analysis, each expression is evaluated by the judgment $(\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D})$. This judgment states that in the universe, Σ, the expression $\Sigma(l)$ is consistently described by the configuration, environment, and dependency sets. For the above judgment to be true, the judgment, $(\Sigma, l) \models (\hat{C}, \hat{\rho})$, must also be true. As with control flow analysis, the dependency judgment is defined differently for each type of syntactical expression. For constants, the judgment is always true. For a variable x labeled l the judgment is satisfied only if $x \in \mathcal{D}(l)$. Trivially, the variable term relies upon its variable. Further definition of dependency analysis may be found in Appendix **??**.

2.3 Transformation

Transformation occurs after the completion of the control flow and dependency analysis. Transformation takes place in three locations: functional terms, application sites, and `let` terms.

At each functional term, the formal parameters that are useless to the body of the function are removed (Wand and Siveroni, 1999). A formal parameter for a functional expression labeled l is useful only if the formal parameter is useful to the body of the function, $DFormals_{(\mathcal{D},\Sigma)}(l_{body})$. (Refer to Appendix **??** for the definition of DFormals.)

At each application site, the actual parameters that are useless to the functions called from this site are removed (Wand and Siveroni, 1999). An actual parameter for an application site labeled l is useful only if it appears in $DActuals_{(\hat{C},\mathcal{D},\Sigma)}(l)$ where DActuals is defined as:

$$DActuals_{(\hat{C},\mathcal{D},\Sigma)}(l) = DFormals_{(\mathcal{D},\Sigma)}(l') \qquad (1)$$

where

$$\Sigma(l) = (t_0^{l_0}\ t_1^{l_1}\ ...t_n^{l_n}) \wedge l' \in \hat{C}(l_0). \qquad (2)$$

The id-val pairs in each `let` term are removed if their respective ids are useless to the body of their respective `let` term (Wand and Siveroni, 1999). If all of the id-val pairs are removed from a `let` expression, the entire `let` construction is replaced by its body.

3 The Creation of UVARP

3.1 Internal Syntax

A procedure was devised to parse the original code into the internal syntax. UVARP converts the supported Scheme expressions into one of eight supported syntactical constructions: `const, var, fn, fun, prim, app, if,` and `let`. These functions closely mimic their Scheme counterparts. Some Scheme constructions were not supported, such as side effects.[2] Table 2 lists the properties of each of the syntactical constructions.[3]

3.2 Discussion of the Structure of UVARP

UVARP was divided into two subprograms: an analyzer and a transformation engine. The analyzer performs both the control flow analysis and the dependency analysis, creating the sets $\hat{C}$, $\hat{\rho}$, and $\mathcal{D}$. The transformation engine outputs the original program as a new program with the useless variables removed.

The first step was to convert the external scheme code into the labeled, lambda calculus. UVARP received the original code as a Scheme list. The list was recursed, adding labels and separating each separate construction into its own sublist. This new code was stored by the expression database, which encapsulated the list of the internal syntax.

After conversion, the analyzer engine was executed. In order to handle a flexible range of syntactical constructions, a procedure was devised to receive the current syntactical symbol and execute the corresponding sub-procedure for each algorithm. These sub-procedures perform 0CFA and dependency analysis, using the solving engine to solve and store the sets, $\hat{C}$, $\hat{\rho}$, and $\mathcal{D}$. The analyzer also uses the expression interface to retrieve the internal syntax from memory. Figure 1 depicts the structure of the analyzer.

The solving engine was divided into two sub-engines: the control flow solver and the dependency solver. The control flow solver generates the sets $\hat{C}_1, ..., \hat{C}_n$ and $\hat{\rho}_{x_1}, ..., \hat{\rho}_{x_m}$, and maintains successor (subset and equal to) relationships between these sets. Similarly, the dependency flow solver generates the sets $\mathcal{D}_1, ..., \mathcal{D}_n$ and maintains all the successor relationships between these sets. The solving engine finds the smallest solution to these successor constraints.

[2] Side effects are commands that require ordered execution i.e. assignments, block structures.

[3] For details, see Wand and Siveroni (1999).

Language	Expression
Scheme	(`lambda` (x) $(add1\ x)$)
Internal Labeled Lambda Calculus	(0 `fn` (x) ((1 `var` $add1$) (2 `var` x)))

Table 1: Translation of a scheme expression into internal-labelled lambda calculus.

Term	Properties
`const` c	A constant, c, such as a number, string, or character
`var` x	A variable, v, whose value is not constant.
`fn` $(v_1, ..., v_n)$ t_{body})	A lambda expression where $v_1, ..., v_n$ are the formal parameters and t_{body} represents the body.
`fun` y $(v_1, ..., v_n)$ t_{body}	A recursive function with formal parameters $v_1, ..., v_n$ and body t_{body}. the fun expression also defines a local name, y.
`prim` $t_1, ..., t_n$	Any undefined function that is assumed to use all its actual arguments, $t_1, ..., t_n$
`app` $t_0, t_1, ..., t_n$	An application site with operator t_0 and operands $t_1, ..., t_n$.
`if` t_{cond} t_{true} t_{false}	A conditional expression. If t_{cond} evaluates to true, then t_{true} is evaluated. Else, t_{false} is evaluated.
`let` $((v_1^{id}\ t_1^{val}), ..., (v_n^{id}\ t_n^{val}))$ t_{body}	A list of local variables, $(v_1^{id}\ t_1^{val}), ..., (v_n^{id}\ t_n^{val})$ defined for the body, t_{body}.

Table 2: Internal Syntax: UVARP parses the functional Scheme code, transforming it into the above syntax.

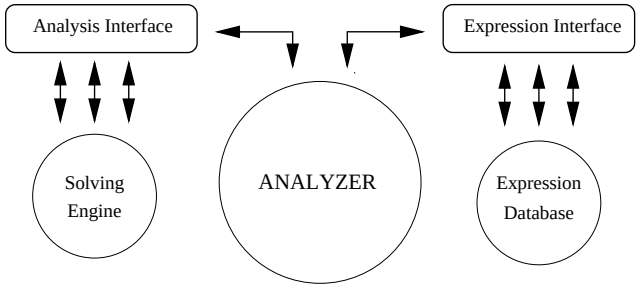

Figure 1: Structure of Analyzer: The arrows represent the data flow. The analyzer accesses the sets $\hat{C}$, $\hat{\rho}$, and $\mathcal{D}$ through the analysis interface. The analysis interface encapsulates the solving engine which continually solves the sets. The expression interface encapsulates the internal code which is stored in the expression database.

During control flow analysis at application site, UVARP must generate successor constraints only if conditional statements of the form, $l \in \hat{C}(l_0)$, become true. If at any point the condition becomes true, UVARP must solve the corresponding successor relationship. These conditional statements posed a problem if l became a member after the condition was initially tested. In order to avoid combinatorial explosion in larger problems due to continual re-testing of the conditional statements, a system was devised in which the conditional statements and their corresponding constraints were stored as procedures with the sets. Anytime a member was added to a set, the procedure would simply check if there were any pending conditions for that member. If at any point the label l was added to the set $\hat{C}(l_0)$, thus satisfying the above hypothetical condition, the stored procedure would be executed. This allowed UVARP to solve the sets synchronous to the execution of the algorithm without actively rechecking past conditions.

Both of the sub-engine solvers access their respective set databases through the graph interface; this interface encapsulates the graph data structure of the sets $\hat{C}$, $\hat{\rho}$, and $\mathcal{D}$. Figure 2 depicts the structure of the solving engine.

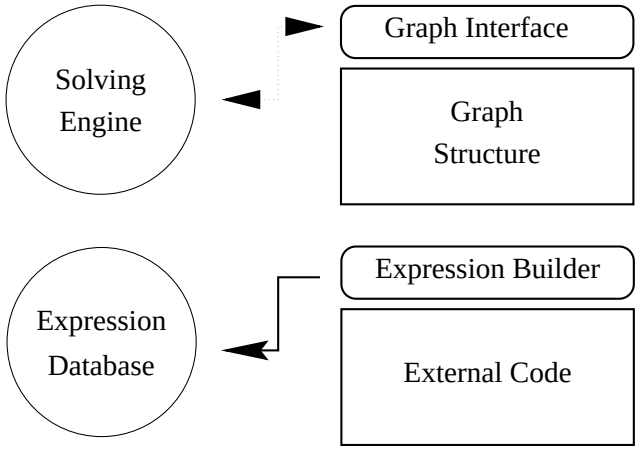

Figure 2: Structure of Solving Engine: The arrows represent the data flow of the solving engine and the expression database. The graph interface encapsulates the graph structure which handles the storage and retrieval of the sets, their members, and their successors. The expression builder converts the input Scheme code into the internal labeled lambda calculus.

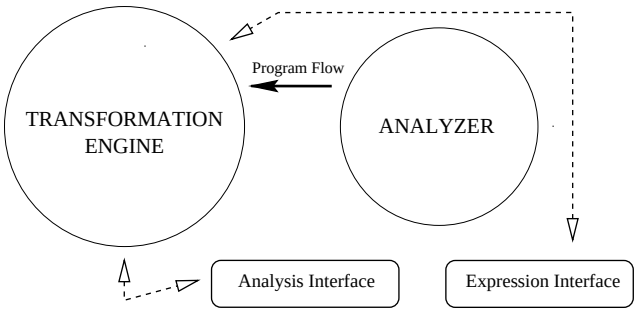

Figure 3: Structure of Transformation Engine: The arrows represent the data flow of the transformation engine. The transformation engine executes after the analyzer completes the control flow and dependency analysis. This transfer in program flow is represented by the single arrow labeled "Program Flow." The transformation engine accesses $\hat{C}$, $\hat{\rho}$, $\mathcal{D}$ through the analysis interface. The transformation engine uses this information to convert the code retrieved through the expression interface.

The transformation engine executes after the analyzer has finished the control flow and dependency analysis. The transformation engine accesses the dependency information through the analysis interface, and accesses the internal code through the expression interface. The internal code is transformed according to the dependency information. The transformed code, stored in external syntax, is returned from the transformation engine. Figure 3 depicts the structure of the transformation engine.

The expression interface was designed to simplify the management of the expression database. The expression interface consists of member functions that encapsulate the underlying expression database; this database, in turn, is built from the external code by the expression builder. The structural overview of UVARP is presented in Figure 4.

3.3 Testing UVARP

After the design and programming were finished, UVARP was tested with many programs. The four most significant are included in this paper to demonstrate the strengths and weaknesses of UVARP. All of the test programs were *specifically engineered to test and illustrate the strengths and weaknesses of the underlying algorithms* in UVARP.

Test 1 - Bogus Variable

```
Original Code:
(let ((loop (fun loop (a bogus j)
       (if (> j 100) a
        (loop a (* bogus 2) (+ 1 j))))))
 (loop b 1 1)) )))

Processed Code:
(let ((loop (fun loop (a j)
      (if (> j 100) a
       (loop a (+ 1 j))))))
 (loop b 1)) )))
```

The first sample program was specifically engineered to test whether UVARP would remove a variable which had no effect on the final computational outcome of the program. UVARP, after processing Test 1, removed the variable, *bogus*. The variable was removed from all formal and actual parameter lists. The program demonstrates UVARP's ability to remove variables that are accessed, updated, and passed between functions, yet do not contribute to the final computational outcome of the program.

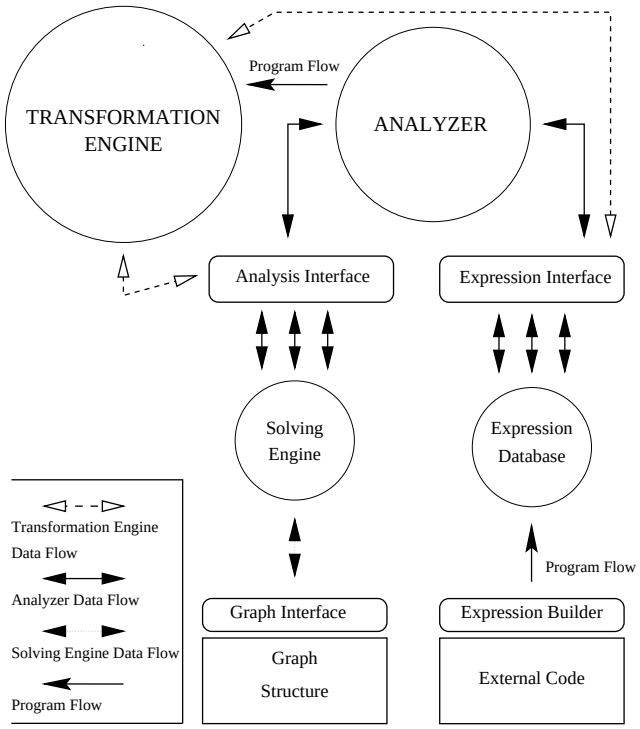

Figure 4: Structure of UVARP: The overall data flow of UVARP is represented by double arrows. The single arrows represent program flow.

Test 2 - Collapsed Let

```
Original Code:
(let ((a 2) (b -1) (c 3)
      (f (lambda (x y z) (- (* 4 (+ x y)) z))))
    ((fun looper (w u l) (if (> w 10) w
      (looper (+ w u) (* u -4) (* l (* u (f a b c))))))
     1 1 1)) ))

Processed Code:
((fun looper (w u)
       (if (> w 10) w
   (looper (+ w u) (* u -4)))) 1 1)
```

The second sample program tested UVARP's ability to completely collapse a `let` expression when all of the `let` expression's local variables (id-val pairs) have no effect on the computational outcome of the program. The code for Test 2 presented UVARP with a `let` expression in which all of the local variables were superfluous. The entire `let` construction was removed along with all instances of the local variables.

Test 3 - Inadvertent Removal of an Infinite Loop

```
Original Code:
(let ((loop (fun loopa (a bogus j)
     (if (> j 100) a
      (loopa (+ a j) (infloop bogus) (+ j 1)))))
  (infloop (fun infloopa (x) (infloopa (+ x 1)))))
    (loop a 1 1)) ))
Processed Code:
(let ([loop (fun loopa (a j)
    (if (> j 100) a (loopa (+ a j) (+ j 1))))])
```

```
(loop a 1))
```

The third sample program originated from Wand and Siveroni's paper on useless variable elimination (Wand and Siveroni, 1999). This problem illustrates a weakness in UVARP's handling of non-termination. The program makes a call to an infinite loop which has no effect on the final computational outcome of the program. UVARP removed the call to the infinite loop. Test 3 reveals an inherent problem in the algorithm. UVARP, when it removes the useless variable $bogus$, eliminates the call to the function $infloop$, thus removing the call to an infinite loop and therefore changing the behavior of the program.[4]

Although the removal of an infinite loop may seem advantageous, this drastic change in program behavior is an unacceptable byproduct of UVARP because the algorithm could create unperceived side effects. One solution would be an extension to the Dependency Analysis that deems any variable useful that contributes to the non-termination of a program. Another solution, though less robust and possibly detrimental to the execution speed, would be to preserve variables whose values contribute to any recursive function.

Test 4 - Constant Value

```
Original Code (same as Processed Code):
(let ((a 2) (b 24) (f (lambda (x y) (+ x y))))
  ((lambda (i j k) (f (+ i (f a b)) (+ j k))) 1 2 3))
```

The fourth sample program illustrates UVARP's inability to remove variables with constant values. The function, f, when called with the variables a and b, will always return 26. From this specific call site, the function f is useless; the constant 26 can be inserted instead of the function call. UVARP returned an identical code to the input code, unable to identify any superfluous variables.

In Test 4, the limitations of 0CFA are revealed. In Test 4, there are two sites from which the function f is called. In first site, $(f\ (+\ i\ (...))\ (+\ j\ k))$, the function f relies on both of its actual parameters because these arguments are not constant. However, the second call site of f, $(f\ a\ b)$, will always return a constant, 26, unless a or b are modified. Therefore, $(f\ a\ b)$ could be replaced with 26. The useless variables $(f\ a\ b)$, a, and b were not removed because 0CFA ignores the calling site when it assess the judgement, $(\Sigma, l) \models (\hat{C}, \hat{\rho})$, for functions.

4 Further Research

UVARP was able to remove the useless variables in the first three test cases but failed to optimize the fourth test case. UVARP finds variables that have no computational effect *throughout the program* and removes the occurrences of these variable within the program. UVARP was unable to remove the useless variables from Test 4 because these variables were not useless to the entire program; the first call site required both its arguments, whereas the second call site could be replaced by a constant. Therefore, UVARP can not remove *intertwined useless variables*, variables whose values contribute to the final computational output yet have only a constant effect or finite set of effects where a simple assignment would replace the variable.

[4] First observed by Wand and Siveroni (1999).

0CFA can not recognize intertwined useless variables because 0CFA neglects function call sites. First-order control flow analysis (1CFA), an extension of 0CFA, generates a set $\hat{C}$ and $\hat{\rho}$ for each possible call site instead of a single $\hat{C}$ and $\hat{\rho}$ for the entire program (Nielson and Nielson, 1997). Accordingly, 2CFA keeps track of a $\hat{C}$ and $\hat{\rho}$ for each call site that may call every other possible call site (Nielson and Nielson, 1997). This process can be repeated infinitely. This is known as Nielson and Nielson's infinitary control flow analysis (Nielson and Nielson, 1997). Infinitary control flow analysis would provide an adequate environment for developing an algorithm that could remove intertwined variables because it evaluates every call site independent of other call sites. This extension to the useless variable elimination algorithm could provide a dramatic increase in execution speed.

5 Conclusion

UVARP reveals only a glimpse of the capabilities of high-level optimizing compilers. The design, programming, and testing of UVARP marks the first implementation of Wand and Siveroni's useless variable elimination algorithm. It can be hypothesized that an implementation of infinitary control flow analysis will increase the optimizing capabilities of UVARP.

6 Acknowledgments

I would like to thank Mitchell Wand of Northeastern University for the time he devoted to me and this project. I would also like to thank Igor Siveroni for his assistance. My thanks to Ramesh Johari, Justin Bernold, and the other tutors at the Research Science Institute for their assistance in revisions. And finally, I would like to thank the Center for Excellence in Education, the Research Science Institute, and Mrs. DiGennaro for providing this research opportunity.

References

S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proceedings 24th Annual ACM Symposium on Principles of Programming Languages*, pages 332–345, January 1997.

O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.

M. Wand. Personal communication, July 1998.

M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–302, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-095-3.

A Control Flow Theory

For any function, `fn`, labeled l and its body labeled l', $(\Sigma, l) \models (\hat{C}, \hat{\rho})$ when $(\Sigma, l') \models (\hat{C}, \hat{\rho})$ and $l \in \hat{C}(l)$. Recursive functions, `fun` $localname$, are identical except for an additional relation: $l \in \hat{\rho}(localname)$.

Primitive functions[5] labeled l of the form $(g^{l_0} t_1^{l_1} ... t_n^{l_n})$ are consistently described by $(\hat{C}, \hat{\rho})$ when their bodies, $t_1^{l_1}, ..., t_n^{l_n}$, are consistently described by $(\hat{C}, \hat{\rho})$. Thus, $(\Sigma, l_i) \models (\hat{C}, \hat{\rho}), i = 1, ..., n$.

`If` expressions of the form $(\texttt{if } t_0^{l_0} \texttt{ then } t_1^{l_1} \texttt{ else } t_2^{l_2})$ labeled l, where $t_n^{l_n}$ represents a term labeled n, are consistently described by $(\hat{C}, \hat{\rho})$ when $(\Sigma, l_i) \models (\hat{C}, \hat{\rho}), i = 1, 2, 3$ and both possible results of the `if` expression, $\hat{C}(l_1)$ and $\hat{C}(l_2)$, are possible results for the entire expression l. Therefore, $\hat{C}(l_1) \cup \hat{C}(l_2) \subseteq \hat{C}(l)$ must be true for $(\Sigma, l) \models (\hat{C}, \hat{\rho})$ to be true.

For any `let` expression labeled l of the form $(\texttt{let } x = t_1^{l_1} \texttt{ in } t_2^{l_2})$, $(\Sigma, l) \models (\hat{C}, \hat{\rho})$ when $(\Sigma, l_i) \models (\hat{C}, \hat{\rho}), i = 1, 2$. Furthermore, every possible value for t_1 becomes a possible value for x, thus $\hat{C}(l_1) \subseteq \hat{\rho}(x)$. Finally, every possible value for the body, t_2, is a possible value for the entire `let` expression, thus $\hat{C}(l_2) \subseteq \hat{C}(l)$.

The last syntactical representation is the `app` labeled l in the form $(t_0^{l_0}\ t_1^{l_1} ... t_n^{l_n})$. Each sub-expression, $t_0 ... t_n$, must be consistently described by $(\hat{C}, \hat{\rho})$. Furthermore, for every member l' of $\hat{C}(l_0)$, where $\Sigma(l')$ is either $(\texttt{fn } (x_1 ... x_n)\ (body_f^{l_f})$ or $(\texttt{fun } y\ (x_1 ... x_n)\ (body_f^{l_f})$, the following relationships are built:

$$\hat{C}(l_i) \subseteq \hat{\rho}(x_i), i = 1, ..., n$$

$$\hat{C}(l_f) \subseteq \hat{C}(l)$$

.

B Dependency Analysis

For every function `fn` labeled l of the form $(\texttt{fn } (x_1 \ ... x_n)\ t_0^{l_0})$, $(\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ when $\Sigma(l_0) \models (\hat{C}, \hat{\rho}, \mathcal{D})$. Since every variable useful to the body except those passed as formal parameters is also useful to the entire function, $\mathcal{D}(l_0) - \{x_1, ..., x_n\} \subseteq \mathcal{D}(l)$.

Similarly, the recursive function `fun` labeled l of the form $(\texttt{fun } localname\ (x_1 ... x_n)\ t_0^{l_0})$ is consistently described by the $(\hat{C}, \hat{\rho}, \mathcal{D})$ when, similar to the `fn` expression, the body, $t_0^{l_0}$, is consistently described by $(\hat{C}, \hat{\rho}, \mathcal{D})$. In addition, all useful variables in the body except those passed as formal parameters to the `fun` expression are also useful to the entire expression. However, the $localname$ variable is useless to the computational value of the entire expression. Thus, $\mathcal{D}(l_0) - \{localname,\ x_1, ..., x_n\} \subseteq \mathcal{D}(l)$.

For every primitive expression labeled l and in the form $(g^{l_0}\ t_1^{l_1} ... t_n^{l_n})$, $(\Sigma, l) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ when $(\Sigma, l_i) \models (\hat{C}, \hat{\rho}, \mathcal{D}) \wedge \mathcal{D}(l_i) \subseteq \mathcal{D}(l), i = 1, ..., n$.

For `if` expressions labeled l in the form $(\texttt{if } t_0^{l_0} \texttt{ then } t_1^{l_1} \texttt{ else } t_2^{l_2})$, $\Sigma(l_i) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ when all terms, $t_i, i = 0, 1, 2$, are consistently described by $(\hat{C}, \hat{\rho}, \mathcal{D})$. Since the outcome of the `if` expression is not known beforehand, all of the variables that are useful in all the terms are also useful to the entire expression. Thus, $\mathcal{D}(l_0) \cup \mathcal{D}(l_1) \cup \mathcal{D}(l_2) \subseteq \mathcal{D}(l)$ must be true for $\Sigma(l_i) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ to be true.

For `let` expressions labeled l in the form $(\texttt{let } x = t_1^{l_1} \texttt{ in } t_2^{l_2})$, the judgment is true when $\Sigma(l_i) \models (\hat{C}, \hat{\rho}, \mathcal{D})$ where $i = 1, 2$. Furthermore, every useful variable in the body except x is possibly useful to the entire expression. Thus, $\mathcal{D}(l_2) - x \subseteq \mathcal{D}(l)$. And finally, if x is possibly useful in the body, then all the useful variables in the expression of $x = t_1^{l_1}$, are possible values for the entire expression. Thus, $x \in \mathcal{D}(l_2) \Rightarrow \mathcal{D}(l_1) \subseteq \mathcal{D}(l)$.

As observed by Wand and Siveroni, if a closure that flows to an application site relies on its i-th parameter, then all closures must also rely on their i-th parameters Wand and Siveroni (1999). These extra conditions can be met by satisfying the following condition:

$$l', l'' \in \hat{C}(l) \Rightarrow DFormals_{(\mathcal{D},\Sigma)}(l') = DFormals_{(\mathcal{D},\Sigma)}(l'')$$

where $\hat{C}(l)$ is the operator of the application site, and $DFormals_{(\mathcal{D},\Sigma)}(l')$ are the useful formal parameters for the term labeled l' with respect to $\mathcal{D}$ and Σ.

Therefore for any application site labeled l of the form $(t_0^{l_0}\ t_1^{l_1}\ ... t_n^{l_n})$ is consistently described by $(\hat{C}, \hat{\rho}, \mathcal{D})$ when, for all the site's operands $t_1^{l_1}, ..., t_n^{l_n}$, $(\Sigma, l_i) \models (\hat{C}, \hat{\rho}, \mathcal{D})$. Furthermore, since all useful variables to the operator are useful variables to the entire expression, $\mathcal{D}(l_0) \subseteq \mathcal{D}(l)$. Finally, for every closure l' that may flow to that site, in the form of $(\texttt{fn } (x_1\ ... x_n)\ t_{body}^{l_{body}})$ or $(\texttt{fun } x\ (x_1\ ... x_n)\ t_{body}^{l_{body}})$, every useful variable to t_{body} is also useful to the application site. Therefore:

$$l' \in \hat{C}(l_0) \wedge x_i \in \mathcal{D}(l_{body}) \Rightarrow \mathcal{D}(l_i) \subseteq \mathcal{D}(l), i = 1, ..., n$$

To force the dependency analysis to maintain the condition,

$$l', l'' \in \hat{C}(l) \Rightarrow DFormals_{(\mathcal{D},\Sigma)}(l') = DFormals_{(\mathcal{D},\Sigma)}(l'')$$

the following condition is added: For all l'' where,

$$\Sigma(l'') = (\texttt{fn } (y_1 ... y_n)\ t_{body'}^{l_{body'}}) \vee (\texttt{fun } y\ (y_1, \ldots, y_n)\ t_{body'}^{l_{body'}}),$$

and $l', l'' \in \hat{C}(l_0)$, useful variables to l_{body} are forced to be useful for $l_{body'}$.

$$\forall l'' : l', l'' \in \hat{C}(l_0) \wedge x_i \in \mathcal{D}(l_{body}) \Rightarrow y_i \in \mathcal{D}(l_{body'}), i = 1, \ldots, n.$$

[5] Primitive functions are undefined functions that require the values of all their formal parameters.